Fourier Volume Rendering on the GPU Using a Split-Stream-FFT

Thomas Jansen, Bartosz von Rymon-Lipinski, Nils Hanssen, Erwin Keeve

Research center caesar
Surgical Systems Lab
Ludwig-Erhard-Allee 2, 53175 Bonn, Germany
Email: {jansen, Lipinski, hanssen, keeve}@caesar.de

Abstract

The Fourier volume rendering technique operates in the frequency domain and creates line integral projections of a 3D scalar field. These projections can be efficiently generated in $O(N^2 \log N)$ time by utilizing the Fourier Slice-Projection theorem. However, until now, the mathematical difficulty of the Fast Fourier Transform prevented acceleration by graphics hardware and therefore limited a wider use of this visualization technique in state-of-the-art applications. In this paper we describe how to utilize current commodity graphics hardware to perform Fourier volume rendering directly on the GPU. We present a novel implementation of the Fast Fourier Transform: This Split-Stream-FFT maps the recursive structure of the FFT to the GPU in an efficient way. Additionally, high-quality resampling within the frequency domain is discussed. Our implementation visualizes large volumetric data set in interactive frame rates on a mid-range computer system.

1 Introduction

Most volume rendering techniques fall into one of two classes:

- In the *screen-space approach*, a ray is cast for each pixel on the screen, with uniform sampling and composition of the volumetric data along the ray, e.g. Raycasting [1] and Shear-Warp [2].
- In the *object-space approach*, the volume is traversed either back-to-front or front-to-back, blending each scalar into the projection plane, e.g. 3D texture mapping [3] and Splatting [4].

It can be seen, that both approaches operate in the spatial domain and somehow have complexity $O(N^3)$ for a volume of size N^3, as each voxel needs to be visited.

Instead of working in the spatial domain, Fourier Volume Rendering (FVR) is based on the frequency spectrum of the 3D scalar field by utilizing the Fourier Slice-Projection theorem. This theorem allows us to compute integrals over volumes by extracting slices from the frequency domain representation.

In detail, FVR generates line integral projections of a set of N^3 scalars – using the inverse 2D Fast Fourier Transform (FFT) – with complexity $O(N^2 \log N)$. The application of the Fourier Projection-Slice theorem to image synthesis has been independently proposed by Dunne et al. [5] and Malzbender [6]. An application to MR angiography is described by Napel et al. [7]. Solutions for depth cueing and illumination where proposed by Totsuka et al. [8] and Levoy [9]. Additionally, frequency domain based algorithms, using the wavelet transform were presented by Westenberg et al. [10] and Gross et al. [11].

However, even the most recent implementations of FVR are realized on the CPU only. In contrast, most spatial domain volume rendering algorithms make use of current graphics hardware features, such as programmability. This leads to faster implementations, even for a worse computational complexity. The mathematical structure of FVR – especially the use of the FFT – has prevented its adaptation to modern graphics hardware.

Such hardware, also known as the Graphics Processing Unit (GPU), nowadays implements a stream architecture, which uses a kernel to operate on one or multiple input streams to produce one or multiple output streams [12]. Using this paradigm, it became popular to use the GPU for problems it was not designed for, e.g. to compute Voronoi diagrams [13], generate interactive caustics [14], or simulate crystal growth [15].

The mathematical obstacle of FVR is the inverse FFT that transforms scalar values from the frequency spectrum to the spatial domain. The adaptation of the recursive structure of the FFT to the GPU is likely to be the major difficulty of the full FVR pipeline. Moreland et al. proposed a FFT implementation [16], however, the algorithm does not make efficient use of the graphics hardware and a special texture format is used, which might be useful for image processing but is not applicable for the FVR approach.

In this paper we describe how to take advantage of the features of commodity graphics hardware to realize FVR. As a major part, this includes a novel implementation of the FFT on the GPU: The *Split-Stream-FFT* was designed to efficiently map the recursive structure of the FFT to the stream architecture of a GPU. This leads to superior speed performance for DSP applications in general, and to the FVR in special. In addition, we deal with resampling in the frequency domain, which is essential to accomplish proper image quality.

2 Methods

Volume visualization can be seen as the inverse problem of tomographic reconstruction. Therewith, it might be useful to take a closer look at it. The objective of tomographic reconstruction is to compute the scalar field $f(x,y,z)$ from a given set of projections. In contrast, in volume rendering the distribution is given and we are asked to compute projections of it.

A common method to achieve tomographic reconstruction is by using the Fourier Projection-Slice theorem [6]. This theorem means for the 2D case, if $F(u,v)$ is the frequency distribution of $f(x,y)$ and $P_\theta(w)$ is the frequency distribution of $p_\theta(r)$, then

$$P_\theta(w) = F(w\cos(\theta), w\sin(\theta)) . \quad (1)$$

Intuitively, the slice of the 2D Fourier transform of an object at some angle θ is the 1D Fourier transform of a projection of the object at the same angle θ (see Figure 1).

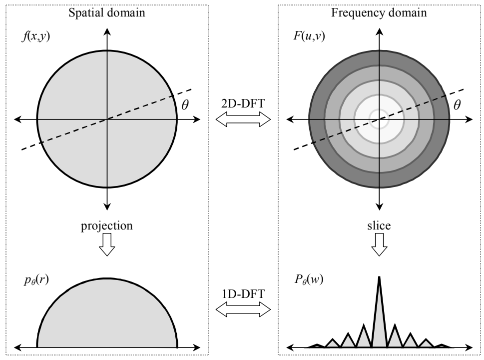


Figure 1: The Fourier Projection-Slice theorem describes the relationship between the Fourier transform of a projection and the Fourier transform of the object.

The Fourier Projection-Slice theorem is still valid in higher dimensions. Starting with a 3D continuous distribution $f(x,y,z)$ and its frequency response $F(u,v,w)$, given by

$$F(u,v,w) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y,z)e^{-2\pi i(xu+yv+zw)}dxdydz \quad (2)$$

a parallel projection of $f(x,y,z)$ can be generated by evaluating $F(u,v,w)$ along a plane that is defined by the orthonormal vectors

$$S = (s_x, s_y, s_z) \quad (3)$$

$$T = (t_x, t_y, t_z) \quad (4)$$

yielding

$$P(s,t) = F(s_x s + t_x t, s_y s + t_y t, s_z s + t_z t) . \quad (5)$$

Taking the inverse 2D Fourier transform leads to

$$p(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} P(s,t)e^{-2\pi i(us+vt)}dsdt . \quad (6)$$

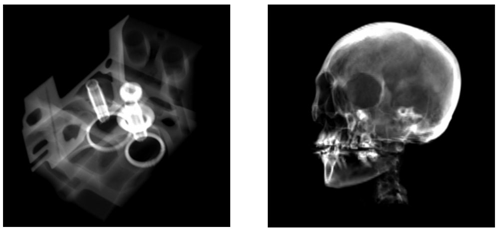
In summary, FVR computes the function

$$p(u,v) = \int_{-\infty}^{\infty} f(t,u,v)dt . \quad (7)$$

Once the forward 3D transform is generated via a pre-processing operation, projections for arbitrary viewing directions can be quickly computed by working with 2D manifolds in the frequency domain. This leads to a better computational complexity and a much better speed performance.

It can be seen that – after the initial pre-processing step – the remainder of the FVR approach can be separated into two distinct parts:

- **High-quality resampling:** The extraction of a plane in the frequency domain, using high-quality interpolation.
- **FFT:** An inverse 2D Fourier transform of the extracted plane yields the X-ray like projection of the volume (see Figure 2).



(a) (b)

Figure 2: Projections generated with our FVR implementation: (a) Engine and (b) CT-Head.

2.1. High-Quality Resampling

Once the volumetric data set is transformed to the frequency domain (pre-processing), we need to extract a plane perpendicular to the view vector. Malzbender has shown that high-quality resampling is more crucial in frequency space than in the spatial domain [6]. Linear interpolation leads to aliasing and ghosting artifacts in FVR.

Generally, there is no direct support in graphics hardware for better interpolation than linear. There are extensions for cubic interpolation, which also might be insufficient or not available at run-time. As a result, we can not rely on the built-in interpolation stage, but need to implement our own interpolation scheme on the GPU.

The implementation of such a scheme is based on the idea of Hart et al. [17]. They realized spline interpolation by exploiting muli-texturing and blending operations. In fact, this leads to the common *Input-Side*-scheme: Instead of traversing through all sample points and gathering the weighted neighbors, this approach traverses through all neighbors and distributes their weighting to the newly generated sample points. The scheme can be extended to an arbitrary neighbor extend in higher dimensions and a custom reconstruction filter. We have chosen the Lanczos windowed-sinc function as the reconstruction filter [18], which is given by

$$\mathrm{Lanczos}_N(x) = \begin{cases} \dfrac{\sin(\pi x)}{\pi x}\dfrac{\sin(\pi x/N)}{\pi x/N}, & |x| < N \\ 0, & |x| \geq N \end{cases} \quad (8)$$

for a neighborhood extend of N.

2.2. The FFT

Fourier analysis is a family of mathematical techniques, based on decomposing signals into sinusoids. The Discrete Fourier Transform (DFT) is used for discrete signals. Such a signal is transformed via the DFT from the spatial domain into the frequency domain. A detailed introduction to Fourier analysis is presented in [19].

Given a sequence of N samples $f(n)$, indexed by $n = 0 \ldots N-1$, the DFT is defined as

$$F(k) = F_N(k,f) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f(n) e^{-2\pi i k n/N} \quad (9)$$

where $k = 0 \ldots N-1$. The inverse DFT is given by

$$f(n) = f_N(n,F) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} F(k) e^{+2\pi i k n/N} \,. \quad (10)$$

There are several ways to calculate the DFT, such as solving simultaneous linear equations or the correlation method. The Fast Fourier Transform (FFT) is a family of other techniques, which is of great importance for a wide variety of applications. The most common FFT technique is the Cooley-Tukey algorithm [20]. This divide-and-conquer approach recursively breaks down a DFT of any composite size $n = n_1 n_2$ into smaller DFTs of sizes n_1 and n_2. The common use is to divide the transform into two pieces of size $n/2$ at each recursion level. The division can take place in the spatial domain (also called time domain) or in the frequency domain, respectively called decimation-in-time (DIT) and decimation-in-frequency (DIF). We have chosen the decimation-in-frequency FFT approach, which is given by

$$F_N(k,f) = \begin{cases} F_{N/2}(\frac{k}{2}, f_e) \text{ for } k = \text{even} \\ F_{N/2}(\frac{k-1}{2}, f_o) \text{ for } k = \text{odd} \end{cases} \quad (11)$$

where

$$\begin{aligned} f_e(n) &= f(n) + f(n+N/2) \\ f_o(n) &= \left(f(n) - f(n+N/2)\right) e^{-2\pi i n/N} \end{aligned} \quad (12)$$

The constant factor $1/\sqrt{N}$ that is part of equations (9) and (10) is discarded in equation (11) in favor of a strict recursive structure. This factor needs to be post-multiplied by $F_N(k,f)$ at the end. The inverse FFT can be computed the same, by changing the *twiddle factor* from $e^{-2\pi in/N}$ to $e^{+2\pi in/N}$. Equations (12) are sometimes referred to as the *FFT butterfly* operation and are graphically shown in Figure 3.

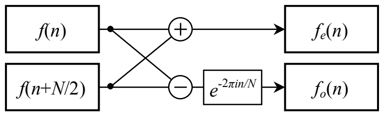


Figure 3: The FFT butterfly is the most essential operation of the Fast Fourier Transform.

Figure 4 illustrates how the algorithm works by showing the naive implementation of the DIF-FFT for $N=8$. Because of the recursive approach, $\log 8 = 3$ stages are used to compute the FFT, each stage performing $8 \cdot 3 = 24$ operations. Generally, the FFT has a complexity of $O(N \log N)$ for N input values. It can be seen, that the resulting frequency distribution is in wrong order. A *bit reversal* (or *tangling*) operation is used to re-sort them at the end. This can be done in constant time. Beside the restrictions of the FVR approach, we have to take care of additional drawbacks introduced by the DFT, e.g. under-/oversampling and aliasing. Details can be found in [19].

3 Implementation

This chapter gives implementation details of the two parts of FVR, pointed out in section 2. We assume that the 3D scalar field was transformed to its frequency representation and transferred to the memory accessible by the GPU.

3.1 High-Quality Resampling

The scheme described in section 2.1 is extended to 3D resampling and implements a Lanczos windowed-sinc function for an extend of N. A naive implementation would lead to $P_N = (2N)^3$ passes, yielding to $P_3 = 216$ for Lanczos_3 and $P_4 = 512$ for Lanczos_4, respectively.
However, a 3D reconstruction filter can be executed by subsequently performing three filters along the major axes. This involves rendering to a 3D render target, which is not possible yet. Fortunately, current GPUs are able to process at least 8 objects at once, therefore, we are able to perform the interpolation for $N \in \{2,3\}$ with rendering to 2D render targets only. This is done as follows (for $N=3$):

1. The 3D input texture is used six times, shifted each time by one voxel along the z-axis. Two weighting textures (for six weights) are used. The fragment program sums up the weighted input voxels and stores the result in $6 \cdot 6 = 36$ intermediate output texture.

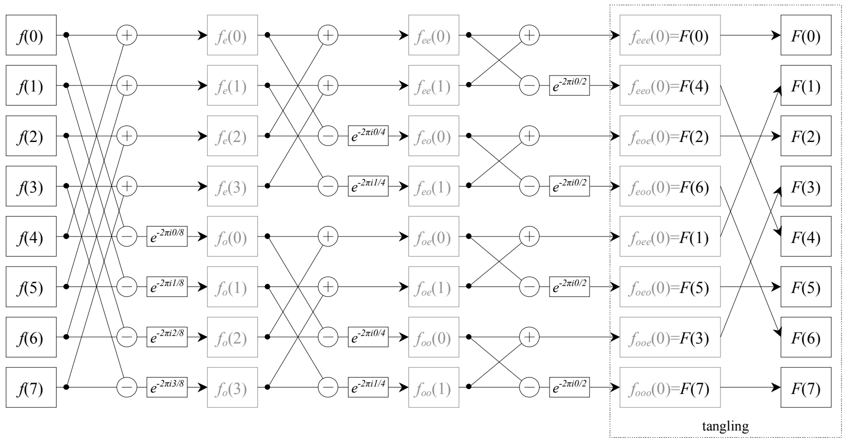


Figure 4: A naive implementation of the DIF-FFT for N=8 scalar values. It can be seen that the frequency distribution needs to be re-sorted (= tangled) at the end, which is a constant time operation.

2. Then, we do the same for the y-axis with the 36 previously generated textures, leading to six new intermediate output textures.
3. At last, these textures are used, each shifted by one pixel along the x-axis, leading to the final texture.

In summary, our implementation requires 36+6+1=43 passes to extract a plane in frequency space for Lanczos_3 and 21 passes for Lanczos_2.

3.2 The FFT

The FFT implementation shown in Figure 4 can not be mapped directly to the GPU. Even if graphics hardware implements a powerful stream architecture, it still is focused on graphics.

3.2.1 Restrictions

We have to deal with the following limitations for each execution of a fragment program:

1. Reading the input stream is optimized for constant step size (called *modulo*), therefore, random access is slow. However, the same input stream can be used multiple times with different offsets and modulos.
2. The output stream is filled subsequently (modulo of 1). Fortunately, the output can begin at any offset, which allows concatenation of outputs.
3. Multiple outputs are allowed for each *fragment program* (kernel), but only one pixel (e.g. RGBA value) per output stream.

Additionally, there are guidelines for the interaction of our FFT fragment programs:

4. The output stream of recursion level n is used as the input stream of recursion level $n+1$. Time-consuming re-ordering of the stream elements should be avoided.
5. Switching the output stream or the fragment program is slow and should be minimized.

3.2.2 Adaptation

As the first step to adapt the DIF-FFT (section 2.2) to the GPU, we take look on restriction 4. Figure 5 presents a reordering of the butterfly operations to achieve consistency between the output of one recursion level and the input of the subsequent level. In addition, the same operation is performed in each recursion level (only twiddle factors are changing). Nevertheless, it is easy to see that other limitations are violated in this configuration.

In the next step, we keep the structure, but reorder the actual output stream elements from back-to-front, see Figure 6. As a result, tangling is shifted to the beginning of the FFT. Most of the restrictions are fulfilled now, except restriction 3.

Therefore, we split the FFT butterfly into two separated passes, each calculating one scalar. The first fragment program is associated to f_e, and the second is related to f_o, respectively (Figure 7).

All restrictions and limitations are satisfied, leading to our *Split-Stream-FFT* implementation. The name is derived from the splitting of the FFT butterfly and the stream re-use between the recursion levels.

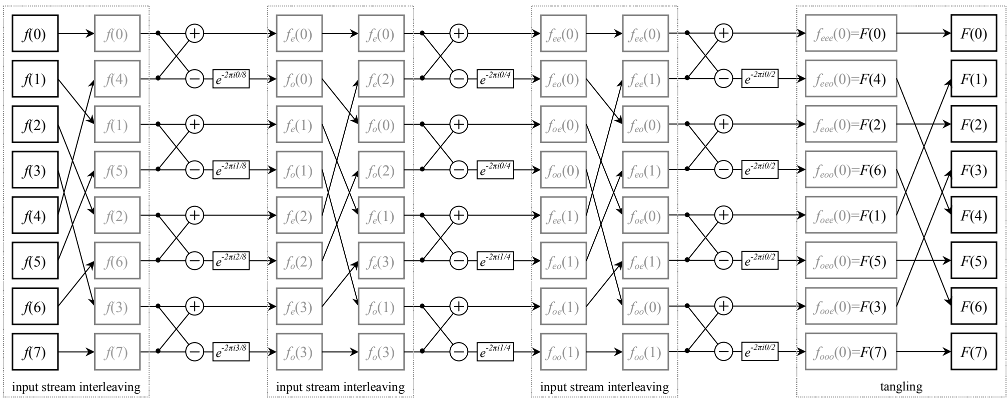


Figure 5: Reordering of the FFT butterfly operations to fit restriction 4. The output stream of one recursion level is the interleaved input stream of the next level. The interleaving can be done for free.

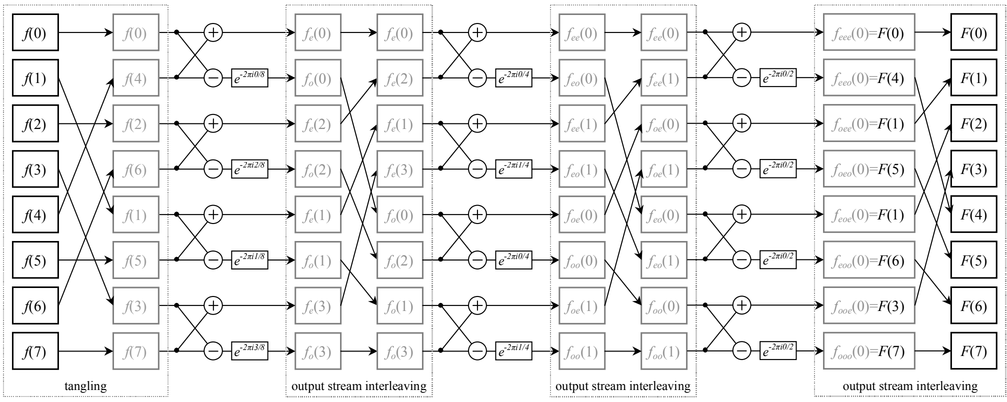


Figure 6: The FFT was re-ordered back-to-front. The tangling has moved to the beginning. Unfortunately, output stream interleaving is not supported by current graphics hardware.

3.2.3 Optimization

Many optimizations are known to speed up a FFT implementation. A technique widely used is working as follows: The last two recursion levels can be combined, due to the simplicity of the twiddle factors used in level $(\log N)-1$

$$e^{-2\pi i 0/4} = (1,0) \text{ and } e^{-2\pi i 1/4} = (0,-1) , \quad (13)$$

and in level $(\log N)$

$$e^{-2\pi i 0/2} = (1,0) . \quad (14)$$

In our implementation, four dedicated fragment programs are used for the last two recursion levels, one for each quarter of the output stream. For small N , we observed a speed-up of about 30%.

The first recursion level is also treated in a special way. As can be seen in Figure 7, the tangling takes place at the beginning. Instead of performing an own *bit-reversal pass*, the first FFT pass is adapted. The input streams are mapped to *texture objects*, therefore the input range can be specified via *texture coordinates*. Instead of supplying the full input stream at once – as for all other recursion levels – we feed many small pieces (of just one value) to the fragment programs of the first recursion level. The order can be controlled by the texture coordinates, achieving a bit-reversal of the input stream. Unfortunately, the cache mechanism of current graphics hardware is not well exploited using this method; therefore, the first recursion level is by far the slowest. This effect can be reduced for 2D FFT, because columns and rows of values can be used.

The Fourier Projection-Slice theorem holds for the Complex-FFT, as well as, for the Real-FFT. Actually, two Real-FFTs can be implemented via a single Complex-FFT, which shrinks the input stream by half and nearly doubling the speed. Details about the simple transformation between Real-FFT and Complex-FFT can be found in [19].

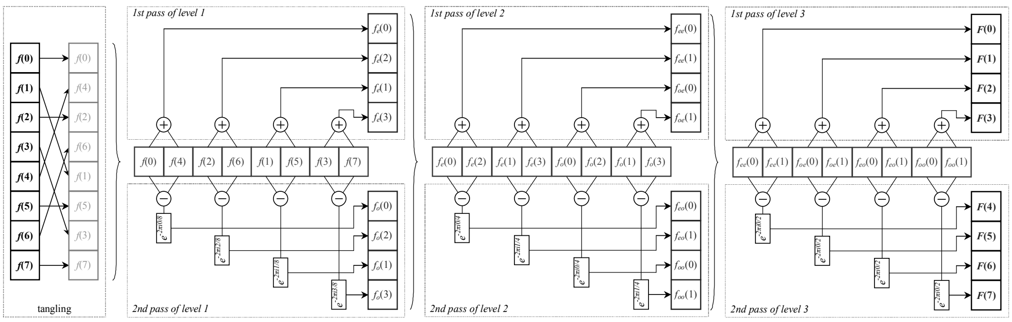


Figure 7: Final implementation of the Split-Stream-FFT. Input streams are interleaved and output streams are concatenated. It is clearly visible, that the FFT butterfly operation is split into two fragment programs.

4 Results and Discussion

In summary, the novelty of our FVR implementation is the FFT, which actually could be used stand-alone by other DSP techniques. Hence, two distinct benchmarks are of interest:

1. **The FFT as a stand-alone operation** compared to third-party implementations. As competitors, we have chosen the latest version (3.0.1, 2003) of the popular FFTW software implementation [21] and the GPU-based adaptation of the FFT approach by Moreland et al. [16]. Performance was measured for a 2D forward FFT of a gray scale image, filtering in frequency space and the 2D inverse FFT. Results of the benchmark can be seen in Table 1.

Table 1: Speed performance (in ms) measured on a 2.6GHz Intel Pentium4 and ATI's Radeon 9800 GPU. (*) Results from Moreland et al. were taken from [16] and downscaled to gray-level image.

Image Size	FFTW	Moreland et al. (*)	Split-Stream
1024^2	535.8	675.0	60.7
512^2	119.9	156.3	14.0
256^2	23.7	37.3	3.4
128^2	1.2	10.0	1.5

2. **The full FVR** compared to other volume rendering techniques. We have chosen a naive ray-casting algorithm in software, as well as hardware accelerated 3D texture mapping. Both competitors use linear interpolation and utilize a saturated addition as the compositing operation. Results of the benchmark are presented in Table 2.

Table 2: Rendering time (in frames per second) for a 512^2 projection of various volumetric data sets. (*) Rate is extrapolated, due to memory limitations on the GPU.

Data Set Size	Raycasting	3D Texture Mapping	FVR on the GPU
512^3	0.3	10.1	14.9 (*)
256^3	0.6	24.2	62.5
128^3	1.3	54.8	143.7
64^3	2.7	121.1	164.8

The dominating argument for FVR is its speed. Most of the other volume rendering techniques have complexity $O(N^3)$ for a volume of size N^3, as each scalar value needs to be evaluated. FVR generates projection images of such volumes with complexity $O(N^2 \log N)$ (excluding the pre-processing step). In fact, the extraction of the projection plane with complexity $O(N^2)$ computationally dominates the inverse 2D FFT for practical values of N. The pre-processing step itself is of complexity $O(N^3 \log N)$.
FVR is limited in additional ways: Because equation (7) is an order independent linear projection along the line of projection t, occlusion is not available. This limits us to transparent visualization, and leads to X-ray like projections of the data set. However, occlusion is not the only depth information. Totsuka et al. address illumination and attenuation and show how this can be implemented directly in frequency space [8]. Other restrictions exist and are discussed in further detail in [5] and [6].
Due to its speed, FVR is practical for large data set visualization. On the other hand, the intermediate representation in frequency space is memory consuming. Instead of 8 or 12 bit per scalar, at least 32 bit for a single-precision floating point value is indispensable. Current consumer graphics hardware is equipped with no more than 256 Mbytes that can hold ideally ~400^3 scalars. However, by quantizing the frequency values, a reduction to 16 bit is possible.

5 Conclusion and Future Work

In this paper, we have presented a novel implementation of the Fast Fourier Transform on the GPU. The Split-Stream-FFT efficiently maps the recursive structure of this fundamental DSP technique to the stream architecture of modern graphics hardware. Additionally, the FFT butterfly operation is split to exploit the rasterization stage of the GPU. By utilizing our approach, volume visualization using the FVR technique has become applicable on commodity graphics hardware. Resampling within frequency space was discussed. Our implementation of the FVR method generates high-quality projections at interactive frame rates.

The work presented in this paper can be enhanced in many ways. We plan to implement the work of Totsuka et al. [8], including other depth cues, e.g. illumination and attenuation. Most of this can be done directly in frequency space, which allows attribute changes (i.e. position of a light source) without performing the pre-processing step. Interactive filtering within the frequency domain might be interesting, as well. This includes smoothing, sharpening and edge enhancement.

Compression is another interesting topic we are working on. Compression in the frequency domain leads to adequate results in quality and compression ratio. This would yield to faster data transfer and support of larger volumes. However, two problems need to be solved: 1) Traditional GPU data structures (e.g. vertex arrays and textures) are inadequate to handle compressed frequency scalars. 2) The extraction stage will increase in complexity.

References

[1] M. Levoy, "*Display of surfaces from volume data*", IEEE Comp. Graph. & Appl., vol. 8, no. 5, 1988.

[2] P. Lacroute and M. Levoy, "*Fast volume rendering using a shear-warp factorization of the viewing transformation*", Proc. SIGGRAPH '94, 1994.

[3] B. Cabral, N. Cam and J. Foran, "*Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*", Proc. Symposium on Volume Visualization '94, 1994.

[4] L. Westover, "*Footprint evaluation for volume rendering*", Proc. SIGGRAPH' 90, 1990.

[5] S. Dunne, S. Napel and B. Rutt, "*Fast Reprojection of Volume Data*", Proc. Conference on Visualization in Biochemical Computing '90, 1990.

[6] T. Malzbender, "*Fourier volume rendering*", ACM Transactions on Graphics, vol. 12, no. 3, 1993.

[7] S. Napel, S. Dunne and B. Rutt, "*Fast Fourier Projection for MR Angiography*", Magnetic Resonance in Medicine, Vol. 19, 1991.

[8] T. Totsuka and M. Levoy, "*Frequency domain volume rendering*", Proc. SIGGRAPH '93, 1993.

[9] M. Levoy, "*Volume Rendering using the Fourier Projection-Slice theorem*", Proc. Graphics Interface '92, 1992.

[10] M. A. Westenberg and J. B. T. M. Roerdink, "*Frequency Domain Volume Rendering by the Wavelet X-Ray Transform*", IEEE Transactions on Image Processing *9(7),* 2000.

[11] M. H. Gross, L. Lippert, R. Dittrich and S. Häring, "*Two Methods for Wavelet-Based Volume Rendering*", Computers & Graphics, 21(2), 1997.

[12] E. Lindholm, M. J. Kilgard and H. Moreton, "*A user-programmable vertex engine*", Proc. SIGGRAPH '01, 2001.

[13] K. E. Hoff, T. Culver, J. Keyser, M. Lin and D. Manocha, "*Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*", Proc. SIGGRAPH '99, 1999.

[14] C. Trendall and A. J. Steward, "*General Calculations using Graphics Hardware, with Applications to Interactive Caustics*", Proc. Eurographics Workshop on Rendering '00, 2000.

[15] T. Kim and M. C. Lin, "*Visual simulation of ice crystal growth*", Proc. ACM SIGGRAPH/EG Symposium on Computer Animation '03, 2003.

[16] K. Moreland and E. Angel, "*The FFT on a GPU*", Proc. SIGGRAPH/EG Conference on Graphics Hardware '03, 2003.

[17] J. C. Hart, "*Perlin Noise Pixel Shaders*", Proc. Eurographics/SIGGRAPH Graphics Hardware Workshop '01, 2001.

[18] K. Turkowski, "*Filters for Common Resampling Tasks*", Graphics Gems I, 1990.

[19] E. O. Brigham, "*The Fast Fourier Transform and Its Applications*", Prentice-Hall, Englewood Cliffs, NJ.

[20] J. W. Cooley and J. W. Tukey, "*An algorithm for the machine calculation of complex Fourier series*", Math. Comput. 19, 1965.

[21] M. Frigo and S. G. Johnson, "*FFTW: An adaptive software architecture for the FFT*", In Proc. Acoustics, Speech, and Signal Processing 3, 1998.